I217: Functional Programming

2. Modules, Order Sorts & Lists of Natural Numbers

Kazuhiro Ogata

Roadmap

- Modules
- Order Sorts
  - Error or Exception Handling
- Lists of Natural Numbers
  - Quicksort
  - Sieve of Eratosthenes
- Exercises

Modules

Units of programs in CafeOBJ. In a module, sorts, operators, variables & equations can be declared. In a module, other existing modules can also be imported (reused).

Declared as follows:

the name of the module

mod! *MOD-NAME* { … }

In the place …, module imports, sorts, operators, variables & equations are declared.

A module declared as this can be reused in other modules like built-in modules, such as NAT.

Modules

```
mod! GCD {
  -- imports
  pr(NAT)
  -- signature
  op gcd : Nat Zero -> Nat .
  op gcd : Nat NzNat -> NzNat .
  op gcd : Nat Nat -> Nat .
  -- CafeOBJ vars
  var X : Nat .
  var NzY : NzNat .
  -- equations
  eq gcd(X,0) = X .
  eq gcd(X,NzY) = gcd(NzY,X rem NzY) .
}
```

The built-in module NAT is imported.

pr stands for protecting, meaning that a module is imported with the protecting mode.

Will you please confirm that if the 2nd argument of gcd is a non-zero natural number, then the result is a non-zero natural number?

Modules

```
open GCD .
  red gcd(24,36) . -- compute the gcd of 24 & 36
  red gcd(2015,31031) . -- compute the gcd of 2015 & 31031
close
```


Modules

```
mod! LCM {
  pr(GCD)
  op lcm : Nat Zero -> Zero .
  op lcm : Nat NzNat -> Nat .
  op lcm : Nat Nat -> Nat .
  var X : Nat .
  var NzY : NzNat .
  eq lcm(X,0) = 0 .
  eq lcm(X,NzY) = (X quo gcd(X,NzY)) * NzY .
}
open LCM .
  red lcm(24,36) . -- compute the lcm of 24 & 36
  red lcm(2015,31031) . -- compute the lcm of 2015 & 31031
close
```

GCD is imported.

If the three **op** declarations for gcd in the module GCD are replaced with the following

op gcd : Nat Nat -> Nat

a warning message on what are called error sorts is displayed when feeding the module LCM into the CafeOBJ system. Why?

Order Sorts

Nat, Zero & NzNat correspond to {0,1,2,...}, {0} & {1,2,…}. As {0} & {1,2,…} are sub-sets of {0,1,2,…}, there are similar relations Zero & NzNat and Nat: Zero & NzNat are *sub-sorts* of Nat (or Nat is a super-sort of Zero & NzNat), which are declared in the built-in module NAT (precisely, in NZNAT-VALUE & NAT-VALUE imported by NAT):

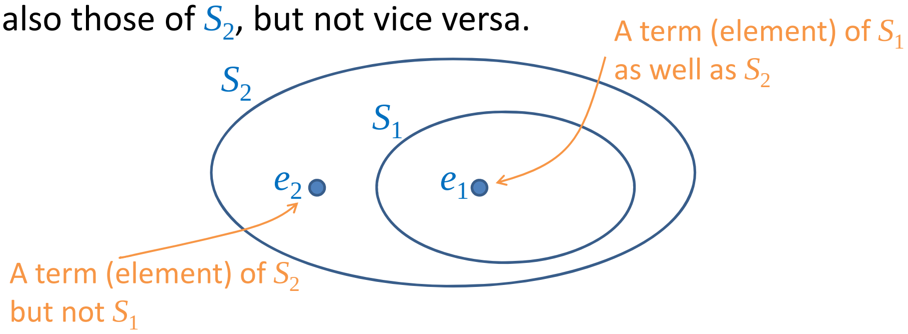
[Zero NzNat < Nat]

As 0 is a member of {0,1,2,…} as well as {0} and 1,2,… are members of {0,1,2,…} as well as {1,2,…}, terms whose sorts are Zero or NzNat are also those of the sort Nat.

Order Sorts

Sub-sort (super-sort) relation is *transitive*: if a sort S_1 is a sub-sort (super-sort) of a sort S_2 and S_2 is a sub-sort (super-sort) of a sort S_3, then S_1 is a sub-sort (super-sort) of S_3.

If a sort S_1 is a sub-sort of a sort S_2, then any terms of S_1 are also those of S_2, but not vice versa.

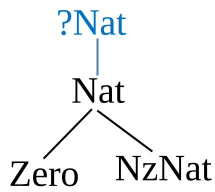


Order Sorts

Suppose that any other sorts have been explicitly declared by users as neither sub- nor super-sorts of any of Nat, Zero and NzNat. Then, Nat, Zero and NzNat forms what is called a *connected component*.

CafeOBJ automatically adds one sort to each connected component such that the sort is a super-sort of all sorts in the connected component and called an *error sort* of the sorts

The sort ?Nat is automatically added as a super-sort of Nat, Zero and NzNat and the error sort of the three sorts.



Order Sorts

Operator _quo_ is declared in the built-in module NAT as follows:

```
op _quo_ : Nat NzNat -> Nat .
```

The result of reducing 1 quo 0 is

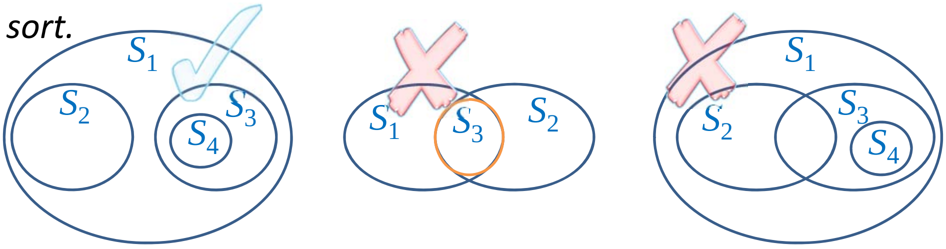
```
(1 quo 0):?Nat
```

which means an error or an exception.

Order Sorts

Only Nat, Zero & NzNat are taken into account. 0 is a term of Nat as well as one of Zero, and 1 is a term of Nat as well as one of NzNat. Zero is the least among Nat & Zero, and NzNat is the least among Nat & NzNat. The *least sort* of 0 is Zero, and the *least sort* of 1 is NzNat.

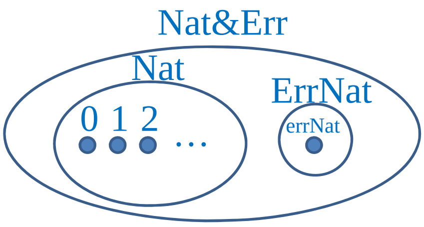
Each connected component of sorts should be designed such that it has the greatest element and each term has the least sort.



Error or Exception Handling

For a connected component *CC* of sorts in which *S* is the greatest element, two new sorts *S&Err* and *ErrS* are added such that *S&Err* is a super-sort of *S* (and then any other sorts in *CC*) and *ErrS* that is neither a sub- nor super-sort of *S* and a constant of *ErrS* is declared.

```
[Nat ErrNat < Nat&Err]    op errNat : -> ErrNat {constr} .
```



It stands for constructor, meaning that errNat is a value of ErrNat.

Note that each connected component of sorts has to have the greatest element.

Error or Exception Handling

Some operators and equations are also declared for error or exception handling.

```
mod! NAT-ERR {  pr(NAT)
 [Nat ErrNat < Nat&Err]
 op errNat : -> ErrNat {constr} .
 op p_ : Zero -> ErrNat .
 op p_ : ErrNat -> ErrNat .
 op p_ : Nat&Err -> Nat&Err .
 op _quo_ : Nat&Err Zero -> ErrNat .
 op _quo_ : Nat&Err ErrNat -> ErrNat .
 op _quo_ : ErrNat Nat&Err -> ErrNat .
 op _quo_ : Nat&Err Nat&Err -> Nat&Err .
 var NE : Nat&Err .
 eq p 0 = errNat .
 eq p errNat = errNat .
 eq NE quo 0 = errNat .
 eq NE quo errNat = errNat .
 eq errNat quo NE = errNat .
}
```


Error or Exception Handling

```
open NAT-ERR .
 red p 1 .
 red p 0 .
 red p errNat .
 red 10 quo 3 .
 red 10 quo 0 .
 red 10 quo errNat .
 red errNat quo 3 .
 red errNat quo errNat .
close
```

What are the results of those reductions?

Lists of Natural Numbers

Collections of natural numbers such that the order is relevant and same numbers can appear multiple times.

Inductively defined as follows:

(1) nil is the empty list of natural numbers.

(2) If *n* is a natural number and *l* is a list of natural numbers, then *n* | *l* is a list of natural numbers such that *n* is the top element of the list.

Why are nil, 0 | nil, 1 | 0 | nil and 2 | 1 | 0 | nil lists of natural numbers?

List of Natural Numbers

Declared in CafeOBJ as follows:

```
mod! NATLIST {  pr(NAT)
  [NatList]
  op nil : -> NatList {constr} .
  op _|_ : Nat NatList -> NatList {constr} .  }
```

constr specifies that the operator is a *constructor*.

Terms without any variables are called *ground terms*.

Ground constructor terms are ground terms constructed from constructors only and interpreted as values.

nil   0 | nil   1 | 0 | nil   2 | 1 | 0 | nil

List of Natural Numbers

What is the top element of a list of natural number?

What if the list is nil?

It totally depends on the definition.

We deal with it as an error, namely errNat.

Partly because of this, two sorts Nil and NnNatList are added as sub-sorts of NatList and the two constructors are revised:

```
mod! NATLIST {  pr(NAT-ERR)
  [Nil NnNatLIst < NatList]
  op nil : -> Nil {constr} .
  op _|_ : Nat NatList -> NnNatList {constr} .  }
```


List of Natural Numbers

The operator hd that basically returns the top element of a given list is declared in the module NATLIST as follows:

```
op hd : Nil -> ErrNat .
op hd : NnNatList -> Nat .
op hd : NatList -> Nat&Err .
```

The equations for hd are declared as follows:

```
eq hd(nil) = errNat .
eq hd(X | L) = X .
```

What are the sorts of hd(nil), hd(0 | nil) and hd(1 | 0 | nil)?

What are the results of reducing hd(nil), hd(0 | nil) and hd(1 | 0 | nil)?

List of Natural Numbers

Some more operators:

```
op tl : NatList -> NatList .
op _@_ : NatList NatList -> NatList .
op [_.._] : Nat Nat -> NatList .
op if_then {_} else {_} : Bool NatList NatList -> NatList .

eq tl(nil) = nil .
eq tl(X | L) = L .
eq nil @ L2 = L2 .
eq (X | L) @ L2 = X | (L @ L2) .
eq [X .. Y] = if X > Y then {nil} else {X | [X + 1 .. Y]} .
eq if true then {L} else {L2} = L .
eq if false then {L} else {L2} = L2 .
```

What are the sorts of hd(tl(nil)), hd(tl(0 | nil)) and hd(tl(1 | 0 | nil))?

What are the results of reducing hd(tl(nil)), hd(tl(0 | nil)) and hd(tl(1 | 0 | nil))?

Quicksort

Invented by C. A. R. Hoare:

C. A. R. Hoare: Algorithm 64: Quicksort. Commun. ACM 4(7): 321 (1961)

Given a list *l* of natural numbers, *l* is partitioned into two lists *ll* and *rl* such that *n* is called a pivot that is a number in *l*, *l'* is *l* from which *n* is deleted, *ll* consists of the numbers in *l'* that are less than *n* and *rl* consists of the other numbers in *l'*, this partition is repeated to *ll* and *rl* until each list obtained by the partition is empty or a singleton, and then all those empty or singleton lists and pivots are combined.

Quicksort

4 | 7 | 5 | 1 | 0 | 3 | 6 | 2 | nil

Based on 4, the list is partitioned.

2 | 3 | 0 | 1 | nil @ (4 | 6 | 5 | 7 | nil)

1 | 0 | nil @ (2 | 3 | nil) 5 | nil @ (6 | 7 | nil)

0 | nil @ (1 | nil)

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | nil

Quicksort

```
mod! QSORT {
  -- imports
  pr(NATLIST)
  -- signature
  op qsort : NatList -> NatList .
  op partition : Nat NatList NatList NatList -> NatList .
  -- CafeOBJ vars
  vars X Y : Nat .
  vars L LL RL : NatList .
```

Quicksort

```
  -- equations
  -- sort
  eq qsort(nil) = nil .
  eq qsort(X | nil) = X | nil .
  eq qsort(X | Y | L) = partition(X,Y | L,nil,nil) .
  -- partition
  eq partition(X,nil,LL,RL) = qsort(LL) @ (X | qsort(RL)) .
  eq partition(X,Y | L,LL,RL)
    = if Y < X then {partition(X,L,Y | LL,RL)}
                else {partition(X,L,LL,Y | RL)} .
}
```


Quicksort

```
open QSORT .
  red qsort(4 | 7 | 5 | 1 | 0 | 3 | 6 | 2 | nil) .
close
```

Sieve of Eratosthenes

An algorithm computing all prime numbers up to a given number.

Given a number n, let l be the list $[2 .. n]$. If l is nil, then nil is the result. Otherwise, let m & l' be $\mathrm{hd}(l)$ & $\mathrm{tl}(l)$, let l'' be l' from which all multiples of m are deleted, l''' be l'' to which the process is recursively applied, and then $m \mid l'''$ is the result.

Sieve of Eratosthenes

2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | nil

2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | nil

3 | 5 | 6 | 7 | 8 | 9 | 10 | nil

3 | 5 | 7 | 8 | 9 | 10 | nil

3 | 5 | 7 | 9 | 10 | nil ➡ 2 | 3 | 5 | 7 | nil

3 | 5 | 7 | 9 | nil

3 | 5 | 7 | 9 | nil

5 | 7 | nil

5 | 7 | nil

7 | nil

Sieve of Eratosthenes

```
mod! ERATOSTHENES-SIEVE {
  -- imports
  pr(NATLIST)
  -- signature
  op primesUpto : Nat -> NatList .
  op sieve : NatList -> NatList .
  op check : Nat NatList -> NatList .
  -- CafeOBJ vars
  vars X Y : Nat .
  var NzX : NzNat .
  var L : NatList .
```


Sieve of Eratosthenes

```
  -- equations
  -- primesUpto
  eq primesUpto(X) = sieve([2 .. X]) .
  -- sieve
  eq sieve(nil) = nil .
  eq sieve(X | L) = X | sieve(check(X,L)) .
  -- check
  eq check(0,L) = L .
  eq check(NzX,nil) = nil .
  eq check(NzX,Y | L)
     = if NzX divides Y then {check(NzX,L)}
                          else {Y | check(NzX,L)} .
}
```

Sieve of Eratosthenes

```
open ERATOSTHENES-SIEVE .
  red primesUpto(10) .
  red primesUpto(20) .
  red primesUpto(50) .
  red primesUpto(100) .
close
```


Exercises

1. Type each module used in the slides and some test code (enclosed with **open** and **close**) in one file and feed it into the CafeOBJ system.

2. Write a module for each piece of programs used in Lecture 1 and some test code in one file and feed it into the CafeOBJ system. Among those modules are FACT & OEDC-FACT.

3. Write a module in which a function that performs the merge sort is defined and some test code in one file and feed it into the CafeOBJ system.

Exercises

4. Write a module in which a function that solves the Hamming's problem is defined and some test code in one file and feed it into the CafeOBJ system. The Hamming's problem is as follows. Given a number n, make the following list of natural numbers.

   (1) Each element of the list is less than or equal to n.

   (2) If the list is not nil, the least element is 1.

   (3) If the list contains x, it also contains $2*x$, $3*x$ and $5*x$.

   (4) Each number occurs in the list at most once.

   (5) The list is in increasing order.